Towards Formal Verification of ToolBus Scripts

Wan Fokkink[2,1] Paul Klint[1,3] Bert Lisser[1] Yaroslav S. Usenko[4,1]

[1] Software Engineering Cluster,
Centrum voor Wiskunde en Informatica,
Amsterdam, The Netherlands

[2] Theoretical Computer Science Section,
Vrije Universiteit Amsterdam, The Netherlands

[3] Programming Research Group,
Universiteit van Amsterdam, The Netherlands

[4] Laboratory for Quality Software (LaQuSo),
Technische Universiteit Eindhoven, The Netherlands

Abstract. ToolBus allows one to connect tools via a software bus. Programming is done using the scripting language Tscript, which is based on the process algebra ACP. Tscript was originally designed to enable formal verification, but this option has so far not been explored in any detail. We present a method for analyzing a Tscript by translating it to the process algebraic language mCRL2, and then applying model checking to verify behavioral properties.

1 Introduction

ToolBus [1, 2] provides a simple, service-oriented view on organizing software systems by separating the *coordination* of software components from the actual *computation* that they perform. It organizes a system along the lines of a programmable software bus. Programming is done using the scripting language Tscript that is based on the process algebra ACP (Algebra of Communicating Processes) [3] and abstract data types. The tools connected to the ToolBus can be written in any language and can run on different machines.

A Tscript can be tested, like any other software system, to observe whether it exhibits the desired behavior. An alternative approach for analyzing communication protocols is model checking, which constitutes an automated check of whether some behavioral property is satisfied. This can be, roughly, a safety property, which must be satisfied throughout any run of the system, or a liveness property, which should eventually be satisfied in any run of the system. To perform model checking, the communication protocol must be specified in some formal language, and the behavioral properties in some temporal logic. Strong points of model checking are that it attempts to performs an exhaustive exploration of the state space of a system, and that it can often be fully automated.

As one of the main aims of Tscript, Bergstra and Klint [2] mention that it should have "a formal basis and can be formally analyzed". The formal basis is

offered by the process algebra ACP, but ways to formally analyze TScripts were lacking so far. This is partly due to a number of obstructions for an automatic translation from TScript to ACP, which are explained below. This work was initiated by the developers of the ToolBus, who are keen to integrate model checking into the design process. This paper constitutes an important step in this direction. We have charted the most important distinctions between ACP and TScript, and investigated how TScript can be translated into the formal modeling language mCRL2 [4]. This language is also based on the process algebra ACP, extended with equational abstract data types [5].

Since both TScript and mCRL2 are based on data terms and ACP, an automated translation is in principle feasible. And as a result, TScript can then be model checked using the mCRL2 or CADP toolset [6]. This method has been applied on a standard example from the ToolBus distribution: a distributed auction. An implementation of an automatic translator from ToolBus to mCRL2 is under development. However, we did have to circumvent several obstructions in the translation from TScript to mCRL2. Firstly, each TScript process has a built-in queue to store incoming messages, which is left implicit in the process description; in mCRL2, all of these queues are specified explicitly as a separate process. Secondly, TScript supports dynamic process creation; in mCRL2, we chose to start with a fixed number of ToolBus processes, and let a master process divide connecting tools over these processes. Thirdly, we expressed the iterative star operator of TScript as a recursive equation in mCRL2. And fourthly, we developed some guidelines on how to deal with so-called result variables in TScript.

Our work has its origins in the formal verification of interface languages [7, 8]. The aim is to get a separation of concerns, in which the (in our case TScript) interfaces that connect software components can be analyzed separately from the components themselves. Our work is closest in spirit to Pipa [9], an interface specification language for an aspect-oriented extension of Java called AspectJ [10]. In [9] it is discussed how one could transform an AspectJ program together with its Pipa specification into a Java program and JML specification, in order to apply existing JML-based tools for verifying AspectJ programs, see also [11]. Diertens [12, 13] uses the ToolBus to implement a platform for simulation and animation of process algebra specifications in the language PSF. In this approach TScript is automatically generated from a PSF specification.

2 ToolBus and Tscript

The behavior of the ToolBus consists of the parallel composition of a variable number of processes. In addition to these processes, a variable number of external tools may be connected to the ToolBus. All interactions between processes and connected tools are controlled by TScripts, which are based on predefined communication primitives. The classical procedure interface (a named procedure with typed arguments and a typed result) is thus replaced by a more general behavior description.

A TScript process is built from the standard process algebraic constructs: atomic actions (including the deadlock `delta` and the internal action `tau`), alternative composition +, sequential composition · and parallel composition ‖. The binary star operation $p * q$ represents zero or more repetitions of p, followed by q. Atomic actions are parametrized with data parameters (see below), and can be provided with a relative or absolute time stamp. A process definition is of the form $Pname(x_1, \ldots, x_n)$ `is` P, with P a TScript process expression and $x_1, \ldots, x_n$ a list of data parameters. Process instances may be created dynamically using the `create` statement.

The following communication primitives are available. A process can send a message (using `snd-msg`), which should be received, synchronously, by another process (using `rec-msg`). Furthermore, a process can send a note (using `snd-note`), which is broadcast to other, interested, processes. A process may `subscribe` and `unsubscribe` to certain notes. The receiving processes read notes asynchronously (using `rec-note`) at a low priority. Processes only receive notes to which they have subscribed. Communication between ToolBus and tools is based on handshaking communication. A process may send messages in several formats to a tool (`snd-eval`, `snd-do`, `snd-ack-event`), and can receive values (`rec-value`) and events (`rec-event`) from a tool.

The only values that can be exchanged between the ToolBus and connected tools are terms of some sort (basic data types booleans, integers, strings and lists). In these terms, two types of variables are distinguished: *value* variables whose value is used in expressions, and *result* variables (written with a question mark) that get a value assigned to them as a result of an action or a process call. Manipulation of data is completely transparent, i.e., data can be received from and sent to tools, but inside ToolBus there are hardly any operations on them. ATerms [14] are used to represent data terms; ATerms support maximal subterm sharing, and use a very concise, binary format. In general, an adapter is needed for each connected tool, to adapt it to the common data representation and message protocols imposed by ToolBus.

The ToolBus was introduced for the implementation of the ASF+SDF Meta-Environment [15, 16] but has been used for the implementation of various other systems as well. The source code and binaries of the ToolBus and related documentation can be found at `www.meta-environment.org`.

3 mCRL2 and CADP

An mCRL2 [4] specification is built from the standard process algebraic constructs: atomic actions (including the deadlock δ and the internal action τ), alternative composition +, sequential composition · and parallel composition ‖. One can define synchronous communication between actions. The following two operators combine data with processes. The sum operator $\sum_{d:D} p(d)$ describes the process that can execute the process $p(d)$ for some value d selected from the sort D. The conditional operator $_ \rightarrow _ \diamond _$ describes the *if-then-else*. The

process $b \rightarrow x \diamond y$ (where b is a boolean) has the behavior of x if b is true and the behavior of y if b is false.

Data elements are terms of some sort. In addition to equational abstract data types, mCRL2 also supports built-in functional data types. Atomic actions are parametrized with data parameters, and can be provided with an absolute time stamp. A process definition is of the form $\mathsf{Pname}(x_1, \ldots, x_n) = P$, with P an mCRL2 process and $x_1, \ldots, x_n$ a list of parameters.

The mCRL2 toolset (`www.mcrl2.org`) supports formal reasoning about systems specified in mCRL2. It is based on term rewriting techniques and on formal transformation of process algebraic and data terms. mCRL2 specifications are first transformed to a linear form [4, Section 5], in a *condition-action-effect* style. The resulting specification can be simulated interactively or automatically, there are a number of symbolic optimization tools, and the corresponding Labeled Transition System (LTS) can be generated. This LTS can, in turn, be minimized modulo a range of behavioral semantics and model checked with the mCRL2 toolset or the CADP toolset [6].

4 From Tscript to mCRL2

Both TSCRIPT and mCRL2 are based on the process algebra ACP [3]. In spite of this common origin, the languages have some important differences, presented later in this section.

Note queues According to the semantics of the TOOLBUS, each process created by TSCRIPT has a queue for incoming notes. A `rec-note` will inspect the note queue of the current process, and if the queue contains a note of a given form, it will remove the note and assign values to variables appearing in its argument list; these can be used later on in the process expression in which the `rec-note` occurs.

mCRL2 contains no built-in primitives for asynchronous communications. Therefore in mCRL2, note queues are handled by a separate AsyncComm process. It also takes care of subscriptions/unsubscriptions and lets any process send any note at any time. Any process can inspect its queue for incoming notes by synchronously communicating with AsyncComm.

Dynamic process creation Process instances may be created dynamically in TSCRIPT using the `create` statement. Although not part of the language, such a process creation mechanism can, in principle, be modeled in mCRL2 using *recursive parallelism*. The latter, however, is not currently supported by the tools in the mCRL2 toolset.

Here, we present a simple solution to this problem, by statically fixing the *maximal* number of process instances that can be active simultaneously. These process instances are present from the start, and the master process divides connecting tools over these processes. To be more precise, for a given TSCRIPT process definition `Pname`, we assume the maximal number of its simultaneously

active instances to be some m. For a translation of `Pname` to an mCRL2 process Pname, we define the following process Pname_inactive,

proc $\mathsf{Pname_inactive}(pid{:}Pid) = \mathsf{r_create}(Pname, pid) \cdot \mathsf{Pname}(pid)$

which after synchronizing with an action s_create proceeds as the process Pname. We instantiate m instances of Pname_inactive in parallel by $\mathsf{Pname_inactive}(1) \parallel \ldots \parallel \mathsf{Pname_inactive}(m)$.

Successful termination of (dynamically) created processes in TSCRIPT is denoted by a `delta` statement. In our approach, the mCRL2 processes do not terminate, but become inactive instead. Therefore, the terminating `delta` statements of `Pname` are translated to $\mathsf{Pname_inactive}(pid)$ recursive calls.

A process willing to create an instance of `Pname` has to execute the mCRL2 expression $\sum_{pid:Pid} \mathsf{s_create}(Pname, pid)$ (instead of a `create` command). As a result of the synchronization with r_create, the creating process gets the *pid* of the "created" process.

Binary star versus recursion TSCRIPT makes use of the binary star operation $p * q$, representing zero or more repetitions of p followed by q. Assuming that the TSCRIPT expression p is translated to an mCRL2 process expression P, and q to Q, the whole TSCRIPT expression $p * q$ is represented in mCRL2 by the recursion variable PQ defined as $\mathsf{PQ} = P \cdot \mathsf{PQ} + Q$.

Local variables TSCRIPT process definitions may make use of local variables and assignments to them. They can be directly translated to process parameters in mCRL2, provided all of them are (made) unique.

Special care has to be taken with the result variables of TSCRIPT, which get a value assigned depending on the context in which they occur. In case they occur in input communication statements like `rec-msg`, `rec-note` or `rec-value`, they can be represented as summations in mCRL2. For example, the TSCRIPT expression `let V:Type in rec-msg(msg(V?))...endlet` can be represented as the mCRL2 expression $\sum_{V:Type} \mathsf{rec_msg}(V) \cdot \ldots$.

In case the result variables occur in process calls, the only way we see to translate them to mCRL2, is to (at least partially) *unfold* the process call instance, so that we get to a situation where the input variable occurs in a communicating statement.

Discrete time One simple option to implement discrete time in mCRL2, is to make use of the tick action synchronization (cf. [17–19]). First, we identify the places where waiting makes sense. These are places where input communication statements are possible. In case no delays are present in these statements, we introduce a possibility to perform the tick action and remain in the current state. In case a TSCRIPT statement is specified with a certain delay, we prepend the resulting mCRL2 translation with the appropriate number of ticks. All TSCRIPT process translations have to synchronize on their tick actions.

Another option is to use the real-time operations built into mCRL2. The current version of the mCRL2 toolset, however, has only limited support for the

analysis of such timed specifications. An interesting possibility is to use clocks to specify timed primitives of TScripts, and to use well-known techniques for analyzing Timed Automata [20] like regions and zones [21] in the context of mCRL2 (cf. [22]).

Unbounded data types A TScript can use variables of unbounded data types, like integers, in communications with the tools. These can be modeled in mCRL2, but the analysis with *explicit-state* model checking techniques will not work. An alternative approach could be in the use of abstract interpretation techniques in the context of mCRL2 (cf. [23]).

5 Conclusion and Future Work

Our general aim is to have a process algebra-based software development environment where both formal verification and production of an executable system is possible. In this paper we looked at a possibility to bring formal verification with mCRL2 to ToolBus scripts.

We presented a translation scheme from TScript to mCRL2. This translation makes it possible to apply formal verification techniques to TScript. We aim at an automated translation tool from TScript to mCRL2, which will make it possible to verify TScript in a fully automated fashion, and to explore behavioral properties of large software systems that have been built with the ToolBus.

The following issues remain as future work.

- Although the translated mCRL2 model is similar in size to the original ToolBus script, its underlying state space may be too large for formal verification. The issues with unbounded data types, timing, and growing note queues due to asynchronous communication, mentioned in Section 4, have to be further addressed.
- The mCRL2 model generated from a particular ToolBus script can be checked for deadlocks, livelocks and some other standard properties. For the analysis of more specific behavioral details one would need properties formulated by the developer of this particular script. Alternatively, a reference mCRL2 model of the tools that communicate with the original script can be considered as an environment for the generated mCRL2 model. Putting this environment model in parallel with the generated mCRL2 model could lead to a more detailed analysis of the external behavior of the original ToolBus script.

References

1. Bergstra, J., Klint, P.: The ToolBus coordination architecture. In Proc. COORDINATION'96, LNCS 1061, Springer (1996) 75–88
2. Bergstra, J., Klint, P.: The discrete time ToolBus - a software coordination architecture. Sci. Comput. Program. **31**(2-3) (1998) 205–229

3. Bergstra, J., Klop, J.W.: Process algebra for synchronous communication. Information and Control **60**(1-3) (1984) 109–137
4. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The formal specification language mCRL2. In: Proc. Methods for Modelling Software Systems. Number 06351 in Dagstuhl Seminar Proceedings (2007)
5. Bergstra, J., Heering, J., Klint, P.: Module algebra. J. ACM **37**(2) (1990) 335–372
6. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In Proc. CAV'07, LNCS 4590, Springer (2007) 158–163
7. Wing, J.: Writing Larch interface language specifications. ACM TOPLAS **9**(1) (1987) 1–24
8. Guaspari, D., Marceau, C., Polak, W.: Formal verification of Ada programs. IEEE Trans. Software Eng. **16**(9) (1990) 1058–1075
9. Zhao, J., Rinard, M.: Pipa: A behavioral interface specification language for AspectJ. In Proc. FASE'03, LNCS 2621, Springer (2003) 150–165
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. In Proc. ECOOP'01, LNCS 2072, Springer (2001) 327–353
11. Larsson, D., Alexandersson, R.: Formal verification of fault tolerance aspects. In Proc. ISSRE'05, IEEE (2005) 279–280
12. Diertens, B.: Simulation and animation of process algebra specifications. Technical Report P9713, University of Amsterdam (1997)
13. Diertens, B.: Software (re-)engineering with PSF III: An IDE for PSF. Technical Report PRG0708, University of Amsterdam (2007)
14. van den Brand, M, de Jong, H., Klint, P., Olivier, P.: Efficient annotated terms. Softw., Pract. Exper. **30**(3) (2000) 259–291
15. Klint, P.: A meta-environment for generating programming environments. ACM TOSEM **2**(2) (1993) 176–201
16. van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Proc. CC'01, LNCS 2027, Springer (2001) 365–370
17. Fokkink, W., Ioustinova, N., Kesseler, E., van de Pol, J., Usenko, Y., Yushtein, Y.: Refinement and verification applied to an in-flight data acquisition unit. In Proc. CONCUR'02, LNCS 2421, Springer (2002) 1–23
18. Blom, S., Ioustinova, N., Sidorova, N.: Timed verification with μCRL. In Proc. PSI'03, LNCS 2890, Springer (2003) 178–192
19. Wijs, A.: Achieving discrete relative timing with untimed process algebra. In Proc. ICECCS'07, IEEE (2007) 35–46
20. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. **126** (1994) 183–235
21. Alur, R.: Timed automata. In Proc. CAV'99, LNCS 1633, Springer (1999) 8–22
22. Groote, J.F., Reniers, M., Usenko, Y.: Time abstraction in timed μCRL a la regions. In Proc. IPDPS'06, IEEE (2006)
23. Valero Espada, M., van de Pol, J.: An abstract interpretation toolkit for μCRL. Formal Methods in System Design **30**(3) (2007) 249–273